Various data can be expressed as tensors
Why do we need to compress tensors?

1. Network I/O
2. Memory requirement

E.g.,) Scientific simulation data
Limitations of existing approaches

- Existing methods heavily rely on assumptions on input data.

Low-rank structure  \( \approx \)  Smooth (e.g., videos)  \( \approx \)  Sparse
Our objective: compression w/o assumptions

• However, not all real-world tensors meet the assumptions.

• How can we compress such general tensors?
Problem definition

Lossy compression of tensors **without** any data assumption.

• **Given**: a general tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times \cdots \times N_d}$.
• **Find**: the compressed data $D$.
• **To minimize**: (1) the size of $D$ and (2) the reconstruction error $\|\mathcal{X} - \mathcal{Y}\|$ where $\mathcal{Y}$ is the tensor reconstructed from $D$. 

$\mathcal{X}$: 90GB

$D$

$\mathcal{Y}$
Outline

1. Introduction.
2. Preliminaries.
3. Proposed method.
4. Experiments.
5. Conclusion.
Our approach is founded on the Tensor-Train decomposition (TTD).

TTD efficiently compresses large matrices.
- E.g., Compression of node embeddings for efficiency of GNNs
Tensor-Train decomposition (TTD)

• TT-cores ($\mathcal{G}$) can be stored instead of the input tensor.
• They can be used to approximately restore the input tensor.
  → lossy compression.
Outline

1. Introduction.

2. Preliminaries.

3. Proposed method.

4. Experiments.

5. Conclusion.
Overview of **TensorCodec**

• Our compression algorithm, **TensorCodec**, makes TTD more expressive, concise, and accurate.

• Q1 Expressiveness: How can we enhance the *expressiveness* of TTD?
• Q2 Conciseness: How can we *reduce the parameters* of TTD?
• Q3 Accuracy: How can we *improve approximation accuracy* of TTD?

• **TensorCodec** employs Neural TTD, Folding, and reordering.
Limited Expressiveness of TTD

• TT-cores are fixed for all tensor entries.

• How can we make TT-cores adaptive to each tensor entry?
A1. Neural TTD (NTTD)

- We make TT-cores **adaptive** to each entry using LSTM returning TT-cores.

```
| Introduction | Preliminaries | Proposed Method | Experiments | Conclusion |
```
A2. Folding

- **Folding** is the process of mapping each entry of a low-order tensor to an entry of a high-order tensor by splitting dimensions.
A2. Folding

• The sum of the mode-sizes of a tensor decreases by folding.

• The **number of parameters of NTTD** is proportional to the sum.

\[16 \times 2 = 32\]

\[4 \times 4 = 16\]
A3. Reordering

Reordering is the process of changing the orders of indices of all modes so that the similar entries are located nearby.
A3. Reordering

- The **closer** the entries are in the original tensor, the **closer they are** in the folded tensor.

- **Reordering** helps the model fit the tensor because they share more inputs to LSTM.
Outputs of TensorCodec

• The outputs of compression are (1) neural-network parameters and (2) an index mapping after reordering.
Reconstruction from the outputs

Entry indices (2, 3, 1) → Reordered indices (12, 23, 2) → Folded indices (2, 4, 6, 3, 2) → Reconstructed value 2.17

- Reordering
- Folding
- NTTD

NTTD?
Summary: contributions of each component

A1. NTTD → Better Expressiveness of TTD

A2. Folding → Better Conciseness of NTTD

A3. Reordering → Better Fitness of NTTD → Better Accuracy
Overall training process for fitting the input

- The outputs of compression are (1) neural-network parameters and (2) an index mapping after reordering.

How to fit?
Overall training process for fitting the input

1. Initialize orders (A3-1).

2. Update NTTD using a gradient descent.

3. Update the orders as in (A3-2).

4. Repeat 2 and 3 until the error converges.
A3-1. Order initialization

• Our goal: minimize the differences between neighboring slices.

• Consider a complete graph.
  • Nodes: slices (i.e., mode indices)
  • Edge weights: L2 distances between the slices.
A3-1. Order initialization

• Find a short cycle with a 2-approximate solution of the TSP.

• Then, remove the largest-weight edge.

• The path becomes the order of slices.
A3-2. Order update using hill climbing

• Finding similar pairs of slices using locality-sensitive hashing (LSH) for L2 distance.

• Swap one slice with the neighboring slice of the other if fitting loss decreases.

• Repeat the above steps.
Outline

1. Introduction.
2. Preliminaries.
3. Proposed method.
4. Experiments.
5. Conclusion.
Experimental settings

• Eight real-world datasets: six 3-order tensors and two 4-order tensors.
Experimental settings

• Lossy-compression baselines:
  • **Low-rank tensor** compression methods
    • CP, Tucker, TT, and TR decompositions.
  • **Smooth-tensor** compression methods
    • TTHRESH and SZ3.
  • **Sparse-tensor** compression methods
    • NeuKron.
TensorCodec is concise and precise

• The compressed outputs of TensorCodec is up to 7.38x smaller.
• TensorCodec shows up to 3.33x better accuracy.
All components of TensorCodec are **useful**

- TensorCodec outperforms all of its variants with missing components.

**TensorCodec (TC)-R:** variant of TC without reordering.

**TC-T:** variant of TC-R without order initialization.

**TC-N:** variant of TC-T without a neural network.
TensorCodec is scalable

• Compression time of TensorCodec is linear in the tensor entry count.
TensorCodec is **scalable**

- Its reconstruction time is **sub-linear** in the tensor entry count.
Further Analysis

• Which slices are *closely* ordered by TensorCodec?

• Can TensorCodec approximate **high-rank tensors** with few parameters?
Reordering by TensorCodec is effective

- Reordered results of TensorCodec align with our intuition.
TensorCodec is expressive

- TensorCodec fits high-rank tensors with a small number of parameters.
Outline

1. Introduction.
2. Preliminaries.
3. Proposed method.
4. Experiments.
5. Conclusion.
Conclusion

• We propose TensorCodec for *lossy compression* of general tensors.
• TensorCodec is *concise*, *accurate*, and *scalable*.
Thank you for listening!
Any question?

Code & Datasets: https://github.com/kbrother/TensorCodec